Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
J. Parallel Distrib. Comput. 68 (2008) 37–53
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Middleware for data mining applications on clusters and grids☆

Leonid Glimcher[a], Ruoming Jin[b], Gagan Agrawal[a,*]

[a]*Department of Computer Science and Engineering, Ohio State University, 2015 Neil Avenue, Columbus, OH 43210, USA*
[b]*Department of Computer Science, Kent State University, Kent, OH 44242, USA*

Received 24 August 2006; received in revised form 9 June 2007; accepted 9 June 2007
Available online 10 July 2007

Abstract

This paper gives an overview of two middleware systems that have been developed over the last 6 years to address the challenges involved in developing parallel and distributed implementations of data mining algorithms. FREERIDE (FRamework for Rapid Implementation of Data mining Engines) focuses on data mining in a cluster environment. FREERIDE is based on the observation that parallel versions of several well-known data mining techniques share a relatively similar structure, and can be parallelized by dividing the data instances (or records or transactions) among the nodes. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. This similarity in the structure can be exploited by the middleware system to execute the data mining tasks efficiently in parallel, starting from a relatively high-level specification of the technique.

To enable processing of data sets stored in *remote* data repositories, we have extended FREERIDE middleware into FREERIDE-G (FRamework for Rapid Implementation of Data mining Engines in Grid). FREERIDE-G supports a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. The added functionality in FREERIDE-G aims at abstracting the details of remote data retrieval, movements, and caching from application developers.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Data mining; Clusters; Grids; Middleware

1. Introduction

Data mining is an inter-disciplinary field, having applications in diverse areas like bioinformatics, medical informatics, scientific data analysis, financial analysis, consumer profiling, etc. In each of these application domains, the amount of data available for analysis has exploded in recent years, making the scalability of data mining implementations a critical factor. To this end, parallel and distributed versions of most of the well-known data mining techniques have been developed. However, we believe that the following challenges still remain in effectively using large data sets and in performing scalable data mining:

Ease of development: Developing efficient parallel applications is a difficult task on today's parallel systems. Clusters of SMPs or multi-core systems, which have been popular or are emerging, offer both distributed memory and shared memory parallelism, which makes application development even harder.

Dealing with large data sets: The data sets available in many application domains, like satellite data processing and medical informatics, easily exceed the total main memory on today's small and medium parallel systems. So, to be scalable to realistic data sets, the parallel versions need to efficiently access disk resident data. Optimizing I/O on parallel configurations is generally harder than on a uniprocessor, which further adds to the complexity of parallel data mining application development.

Maintaining and performance tuning parallel versions: Maintaining, debugging, and performance tuning a parallel application is an extremely time consuming task. As parallel architectures evolve, or architectural parameters change, it is not easy to modify existing codes to achieve high performance on new systems. As new I/O, communication, and synchronization optimizations are developed, it is useful to be able to apply them to different parallel applications.

☆This research was supported by NSF Grants #CNS-0203846, #CCF-0541058, and CNS #0403342.

*Corresponding author. Fax: +1 614 292 2911.
E-mail addresses: glimcher@cse.ohio-state.edu (L. Glimcher), jin@cs.kent.edu (R. Jin), agrawal@cse.ohio-state.edu (G. Agrawal).

0743-7315/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2007.06.007

Currently, this cannot be done for parallel data mining implementations without a high programming effort.

Support for processing remote data sets: Analysis of large geographically distributed scientific data sets, also referred to as *distributed data-intensive science* [10], has emerged as an important area in recent years. Scientific discoveries are increasingly being facilitated by analysis of very large data sets distributed in wide area environments. Careful coordination of storage, computing, and networking resources is required for efficiently analyzing these data sets. Even if all data are available at a single repository, it is not possible to perform all analysis at the site hosting such a shared repository. Networking and storage limitations make it impossible to down-load all data at a single site before processing. Thus, an application that processes data from a remote repository needs to be broken into several stages, including a data retrieval task at the data repository, a data movement task, and a data processing task at a computing site. Because of the volume of data that is involved and the amount of processing, it is desirable that both the data repository and computing site may be clusters. This can further complicate the development of such data processing applications.

This paper gives an overview of two middleware systems that have been developed over the last 6 years to address the above challenges. FREERIDE (framework for rapid implementation of data mining engines) focuses on data mining in a cluster environment. FREERIDE is based on the observation that parallel versions of several well-known data mining techniques share a relatively similar structure. We have carefully studied parallel versions of a priori association mining [1], bayesian network for classification [8], k-means clustering [22], k-nearest neighbor classifier [19], and artificial neural networks [19]. In each of these methods, parallelization can be done by dividing the data instances (or records or transactions) among the nodes. The computation on each node involves reading the data instances in an arbitrary order, processing each data instance, and performing a *local reduction*. The reduction involves only commutative and associative operations, which means the result is independent of the order in which the data instances are processed. After the local reduction on each node, a *global reduction* is performed. This similarity in the structure can be exploited by the middleware system to execute the data mining tasks efficiently in parallel, starting from a relatively high-level specification of the technique.

To enable processing of data sets stored in *remote* data repositories, we have extended FREERIDE middleware into FREERIDE-G (FRamework for Rapid Implementation of Data mining Engines in Grid). FREERIDE-G supports a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. The added functionality in FREERIDE-G aims at abstracting the details of remote data retrieval, movements, and caching from application developers.

This paper also presents a subset of application development efforts and experimental results we have obtained from these two systems. Specifically, we describe our experience in developing a molecular defect detection application on FREERIDE. We also present initial performance evaluation of FREERIDE-G using three data mining algorithms and two scientific data processing applications.

The rest of this paper is organized as follows. An overview of FREERIDE is presented in Section 2. Molecular defect detection case study is discussed in Section 3. FREERIDE-G design described in Section 4 and experimentally evaluated in Section 5. We given an overview of related research efforts in Section 6 and conclude in Section 7.

2. FREERIDE middleware

In this section, we describe the basic functionality and interface of our FREERIDE middleware.

FREERIDE is a general framework for parallelizing data mining algorithms on both distributed and shared memory configurations. It also provides support for efficient execution on disk-resident data sets. In the past, FREERIDE has been used for a number of well-known data mining algorithms, including a priori and FP-tree based association mining, k-means and EM clustering, decision tree construction and nearest neighbor searches. The details of the functionality and results from evaluation of the system are available in our earlier publications [24–29].

FREERIDE is based on the observation that a number of popular data mining algorithms share a relatively similar structure. Their common processing structure is essentially that of *generalized reductions*. During each *phase* of the algorithm, the computation involves reading the data instances in an arbitrary order, processing each data instance, and updating elements of a *reduction object* using associative and commutative operators.

In a distributed memory setting, such algorithms can be parallelized by dividing the data items among the processors and replicating the reduction object. Each node can process the data items it owns to perform a local reduction. After local reduction on all processors, a global reduction can be performed. In a shared memory setting, parallelization can be done by assigning different data items to different threads. The main challenge in maintaining the correctness is avoiding race conditions when different threads may be trying to update the same element of the reduction object. We have developed a number of techniques for avoiding such race conditions, particularly focusing on the impact of locking on memory hierarchy. However, if the size of the reduction object is relatively small, race conditions can be avoided by simply replicating the reduction object.

A particular feature of the system is the support for efficiently processing disk-resident data sets. This is done by dividing the data set into a set of *chunks*. Then, the processing time is minimized by reading the chunks in an order that minimizes the disk seek time, and aggressively using asynchronous read operations.

Since our main focus is on parallelization in a distributed memory environment and scaling to disk-resident data sets, we describe the interface available for facilitating these. The following functions need to be written by the application developer using our middleware.

The subset of data to be processed: In many cases, only a subset of the available data needs to be analyzed for a given data mining task. These can be specified as part of this function.

Local reductions: The data instances or chunks owned by a processor and belonging to the subset specified are read. A local reduction function specifies how, after processing one chunk, a *reduction object* (declared by the programmer), is updated. The result of this processing must be independent of the order in which the chunks are processed on each processor.

Global reductions: The reduction objects on all processors are combined using a global reduction function.

Iterator: A parallel data mining application often comprises of one or more distinct pairs of local and global reduction functions, which may be invoked in an iterative fashion. An iterator function specifies a loop which is initiated after the initial processing and invokes local and global reduction functions.

3. A detailed case study using FREERIDE

This section presents a case study in creating a parallel and scalable implementation of a scientific data analysis application using FREERIDE. We focus on a defect detection and categorization application [34]. This application analyzes data sets produced by molecular dynamics (MD) simulations, which comprise locations of the atoms and the associated physical quantities. The goal of the analysis is to detect and categorize the defects in the data sets. Because of the presence of noise in these simulations, it is important to classify the identified defects into similar classes. Thus, this application involves two major phases. In the *defect detection* phase, atoms are marked as defect atoms based on statistical rules and then clustered to form defect structures. In the *defect categorization* phase, we use a shape matching algorithm to try and match each defect to an existing defect catalog.

In parallelizing this application, we had the following three goals. First, we obviously wanted to achieve high parallel efficiency. Second, we wanted to create an implementation that can scale to disk-resident data sets. Finally, we wanted to create an easy to maintain and modify implementation, which is possible only through using high-level interfaces.

3.1. Feature based mining approach

Direct numerical simulations are being increasingly used to study many physical, chemical and biological phenomena. An important goal of MD simulations is to uncover fundamental defect nucleation and growth processes in Silicon (Si) lattices, either in the presence of thermal sources or extra atoms (e.g., additional Si atoms or dopants such as Boron). These defects can alter electrical and material properties (e.g., capacitance) of semi-conductor devices dramatically. Therefore, to precisely control the mechanisms of device fabrication, it is important to understand the extent and evolution of the defect structures.

The challenges in detecting defects and classifying them from data produced by an MD simulation are daunting. The data sets produced by MD simulation are often very large. The output is comprised of the locations of the atoms and associated physical quantities, including energy and potential. In typical Si defect simulations, more than 10 million time steps are generated to study the evolution of single- or multi-interstitial in a lattice. Manual analysis to seek and classify individual defects is both cumbersome and error-prone. Therefore, there is a need to develop fast automatic detection and classification schemes that scale well to increasingly large lattice systems.

A detection and categorization framework has been developed to address the above need. It consists of the two phases with several sub-steps in each phase. We next briefly summarize both phases with the associated sub-steps. A more detailed overview of the approach is available in a recent publication [34].

3.1.1. Phase 1-defect detection

In this phase the atoms are marked as defect atoms based on statistical rules and then clustered to form one or more defect structures.

Local operators: The local operators (rules) check each atoms for correct number of neighbors and bond angles. All the atoms which do not follow these rules are marked as defect atoms. For Silicon lattice the number of neighboring atoms should be 4 and each dihedral angle should be $\in [90°,130°]$. Two atoms are neighboring atoms if the euclidean distance between them is $\leqslant 2.6$ Å.

A bulk silicon atom has precisely four neighbors within the distance of 2.6 Å and the angles between any two bonds lie within [90°,130°]. Any other atom is a defect. Similar definitions can be formulated for other systems. In a solid, the periodic boundary condition has to be treated with care to obtain the correct bond lengths and distances near the boundary.

Clustering the marked atom: Segmentation of defects is performed through aggregating defect atoms in one or more connected sub-structures (defects). A line is drawn connecting all defect atoms that lie within a distance of 4 Å of each other. Each cluster is then a connected graph, which is computationally inexpensive to obtain given the relatively small number of atoms in a defect. Fig. 1 shows two defects embedded in a 512 atom lattice. The different shades represent distinct and separated spatial clusters (defects) in a 512-atom Si lattice.

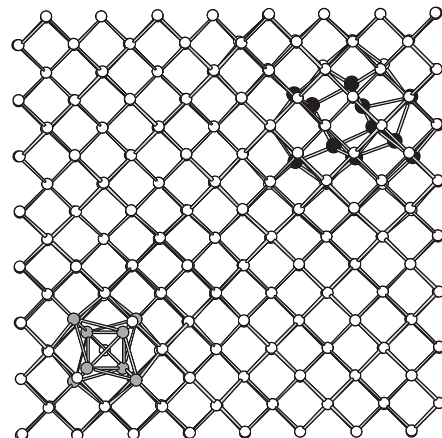

Fig. 1. Lattice with two detected defects.

3.1.2. Phase 2-defect categorization

This phase consist of two substeps. The first step, which is computationally inexpensive, provides with a set of candidate defect classes. The second step tries to match the candidate classes from first step using a relatively expensive exact shape matching algorithm.

Pruning using feature vectors: The shape of the defect is well represented by using central moment till third order. A k-nearest neighbor classifier is then used to pick the closest K classes. These K classes are the input to next step.

Exact match using largest common sub-structures: This step finds the largest common sub-structure (LCS) between the defect to be classified and candidate classes. The LCS is found by using distance and orientation of atoms. The class which gives largest size LCS is considered the class of the defect. However, if the number of atoms in LCS is $\leqslant M$ (a user specified threshold), then the defect is considered to be new and the moment vector and positions of atoms are added to the database.

3.2. Parallel algorithm and implementation

We now describe a number of algorithmic and implementation issues in parallelizing the defect detection and categorization application described in the previous section.

Initially, we discuss the key ideas involved in our implementation. Then, we describe how we use FREERIDE as the middleware for parallelization. Finally, we describe the sequence of steps involved in the parallel implementation.

3.2.1. Key issues

The key issues in our parallel implementation are as follows. First, we wanted to have an implementation which can scale to disk-resident data sets easily. This requires organizing and processing input data as chunks, and not assuming that the data set can fit in main memory. The second important consideration was achieving good load balance, which can be challenging for the defect categorization phase. Finally, we had to avoid sequentializing any computation, which again was challenging for the defect matching phase.

Data organization and chunk boundary replication: Partitioning of the data set into chunks is a necessity for scaling the processing on disk-resident data sets. The input grid is partitioned in the 3-D space and cubical sections of a uniform size are assigned to a chunk. The grid points corresponding to a chunk are stored contiguously. In a parallel setup, an equal number of chunks are assigned to each processor.

One particular issue in data organization came up because of the need to avoid communication during the binary classification phase. As we had discussed earlier, for binary classification of each atom the bond lengths to all neighbors in space are required. This poses a challenge for processing the atoms located at any of the surfaces of each chunk. A naive approach would require a round of communication to perform classification of the boundary atoms. Moreover, even on each node, the lattice atoms within a chunk cannot be processed independently.

The above problems can be addressed by replicating the atoms that are on any of the surfaces of a chunk. Such surface atoms are now included as part of the original chunk, as well as its neighbor.

The silicon lattice is partitioned in the following way: atoms with minimum and maximum coordinates in a 3-D space make up corners of a rectangular box, inside which the lattice would fully fit. Instead of partitioning the lattice, we now partition its container: for every time each of the three dimensions is split into 2^i parts the number of chunks increases by a factor of 2^{3i}. All atoms that are located within four bond lengths of the chunk boundary are replicated as a part of data belonging to both chunks that share the boundary.

Achieving good load balance: Good load balance is essential for any parallel implementation. As we will demonstrate in the experimental results section, both defect detection and defect categorization are computationally demanding. Thus, achieving good parallel performance for both of these phases is critical to our implementation.

Achieving good load balance for the detection phase is relatively easy. The lattice is partitioned into chunks of equal size, and the same number of such chunks is assigned to each processing node. Moreover, it turns out that each lattice chunk is almost equally likely to contain a defect, so the work is split up evenly between the processing nodes.

To achieve good load balance in the categorization phase, the number of defects that each node classifies should be roughly the same, or, at least, no single node should have to do significantly more work than the other processing nodes. There are two ways defects can be identified in our parallel implementation. The first possibility is that a defect is *local* to a node, i.e., the corresponding atoms were on one or more chunks on a single node. The second possibility is that the corresponding atoms spanned multiple processing nodes. In this case, the nodes send incomplete defects to a master node, which then completes the growth of the defect.

A naive implementation will be to have the defects of the second type categorized on the master node. The rest of the defects, then, would be categorized on whichever node they belong to. This, however, would assign significantly more work to the master node, and would result in poor load balance. We avoid this problem with a simple heuristic that was easy to implement and did not introduce any significant communication overhead.

When each node sends its set of incomplete defects, it also sends the number of complete defects it has. Let the number of complete defects on a node i be $m(i)$. Let the number of defects identified on the master node using two or more incomplete defects be n and let the number of nodes be p. We do the following calculations:

$$T = \sum_i m(i) + n,$$

$$A(i) = \max(T/p - m(i), 0),$$

$$D = \sum_i A(i) - n.$$

T is the total number of defects in the system. Ideally, we will like to have T/p defects for classification on each node. If no node has more T/p defects identified locally, we can assign $T/p - m(i)$ defects from the master node to the node i. However, this may not always be true. In such cases, D is the deficit on the number of defects to be reassigned, which must be evenly balanced among all nodes being reassigned defects.

This simple scheme avoid expensive communication for achieving perfect load balance, and works quite well in practice.

Categorizing non-matching defects: Categorization is performed by comparing a moment vector of a defect with mean moment vector for defect classes available from the defect catalog. But what happens when a certain class is not represented in the catalog? This usually means that the catalog is incomplete, and that it needs to be updated to include mean moment vectors for the non-matching defect. Once the catalog is updated, we need to use the new catalog for further matches.

The need for updating and maintaining a consistent catalog creates a problem in correct parallelization. One approach will be to perform categorization of non-matching defects on the master node. However, this requires that part of the work be sequentialized. In the worst case, the database could be empty initially and every defect encountered could be new, which will result in all of categorization phase being sequentialized.

Our implementation uses a better approach. Each processing node adds each non-matching defect it encounters to its own private copy of the database under a temporary name. This step ensures that all of the defects that are of the same class and are to be processed after the current one will match the database, and will be assigned this temporary class name. Each processing node also keeps a separate record of all the new classes that it encountered while performing the categorization phase. This collection of new classes from each processing node is then sent to the master node, where the duplicates across the nodes are identified. Then, each class with a temporary name is assigned a new name, and these names are broadcasted to all nodes. Each node then replaces the temporarily assigned class names with the new names.

3.2.2. Using FREERIDE

Our parallel implementation was carried out using a cluster middleware called FREERIDE.

The FREERIDE system and its programming interface can easily allow the following steps: (1) retrieval of chunks of interest on each node, (2) *local* processing over the data elements comprising a chunk, (3) processing on individual nodes after all chunks of interest have been processed, (4) a *global combination step*, and (5) postprocessing on one or more nodes.

The above five steps can be repeated multiple times, which is required in order to split up the defect detection and the defect categorization phases between iterations of the parallel application. Consider the steps of the defect detection and categorization framework presented in the previous section. The analysis is usually performed on the entire lattice, which means all chunks resident on a node need to be retrieved and processed in the detection phase.

The first step, i.e. rule discovery, involves calculating bond lengths and angle between an atom and each of its neighbors in the lattice. For a silicon lattice, every atom that forms a number of bonds other than 4, or whose bond angles are outside a specified range are classified as defects. Because of the boundary replication that we described earlier, this can be done easily as a part of local processing.

The second step, i.e. segmentation of defects, is the more involved of the two steps in this phase. When the neighboring atoms classified as defects are within a chunk, this segmentation can be done as a part of local processing on the chunk. However, one defect can easily extend across multiple chunks or even nodes. Thus, this segmentation step needs to be performed through a combination of local processing on each chunk, the processing step after all chunks on a node have been processed, and the global combination step. Further details of this will be the core of Section 3.2.3.

The first step of the classification phase, i.e. pruning, is (like rule discovery) purely local, except for the need for load balancing that we described earlier. The second step of classification, i.e. matching, is more complex. As we described earlier, if a defect under question matches the entry in the database, LCS matching can be carried out as a part of local processing. However, if a defect does not match the database and needs to be added to it, then the LCS matching step needs to be carried out as a combination of both local and global processing steps.

Overall, the structure of our target application matches the processing structure of the FREERIDE system. In the next subsection, we give full details of the parallel algorithm.

3.2.3. Parallel algorithm description

This subsection gives a more detailed description of the parallel algorithm and its implementation. The implementation broadly consists of seven steps, which are:

1. Detecting defects on each chunk and classification of defects as *complete* or *incomplete*. The defects are stored as a part of the *reduction object*.
2. Combining or growing *incomplete* defects detected across different chunks belonging to each node. At the end of this process, the defects are again classified as *complete* or *incomplete*.
3. Communicating information required for further defect growth to a master node.
4. Growing incomplete defects from all nodes.
5. Redistributing defects to processing nodes for the categorization phase. Performing pruning and LCS matching. *Matching* defects are classified in this step and need no further processing. *Non-matching* defects are marked for further processing and assigned to temporary classes.
6. Communicating *non-matching* defects to the master node. These are representatives of the new defects to be added to the defect catalog. Matching *new* defects from all nodes against each other to get rid of duplicates. Resulting list is used to update the catalog.

7. The master node broadcasts the new class names back to all processing nodes. These names are used to finalize classification of the *non-matching* defects on all nodes.

The rest of this subsection describes each of the above steps in more details.

Step 1: Binary classification and intra-chunk region growth. Deciding whether or not a specific lattice atom belongs to a defect depends on the number of bonds and the size of bond angles that each atom forms with its neighbors. Once the surface points have been replicated across the chunks, this step is quite simple. After the detection and binary classification, the aggregation step is initiated within the chunk. In the original sequential algorithm, the aggregation step simply involves finding a grid point that is classified as being part of a defect, and then continuing to add its immediate neighbors that meet the same criteria. As we described earlier, when applying this step within a chunk, there are two possibilities when we find that a defect cannot be grown any further, which correspond to having complete and incomplete defects, respectively.

One of the challenges is how we can store information about incomplete chunks and facilitate growing them using points from other chunks. We store what we refer to as *face imprints*. Up to six face imprints can be stored for each incomplete defect, one for each of the surfaces of the chunk. For each surface, a face imprint simply stores the points belonging to the defect that are on a surface of the chunk.

Step 2: Intra-node aggregation. After processing each chunk independently, an *intra-node* aggregation step tries to grow each defect that was incomplete at the end of the previous step. This step can be performed by using only the reduction object, where the face imprints of incomplete defects from all chunks are stored. The entire chunks do not need to be read again. We assume that the face imprints of incomplete defects of all chunks can be easily stored in main memory. The intra-node aggregation phase involves several steps, as listed below:

1. *Coordinate mapping*: For aggregating defects across chunks, we need to have a consistent coordinate system across different chunks. Because the coordinates of atoms are not explicitly stored, the only identifier of a point within a chunk is its offset. Using the location of chunk in the original grid and the offset within the chunk, we need to compute the position of a point within the entire grid.

2. *Defect growing*: Consider any incomplete defect. For each of its face imprints, we find the chunk that has the adjacent surface. We first check if this chunk has an incomplete defect and that defect has a face imprint on that surface. If so, we compare the two face imprints. If one or more points from the first face imprint neighbors a point from the second face imprint, the two defects can be merged. This process is repeated till no two defects can be merged any further. By careful organization, the above can be carried out using only a single pass on the set of chunks within the node.

3. *Creating new data-structures*: At the end of the intra-node defect growing phase, we determine the new set of complete and incomplete defects.

Step 3: Inter-process communication. After the end of local processing, a communication step is carried out. One important question is, what data structures need to be communicated. One possibility is to communicate all points from all incomplete defects to a single node and then try to grow them further. However, this can be unnecessarily expensive. The face imprints of all incomplete defects are sufficient to determine which defects could be merged. Therefore, our implementation is limited to communicating the face imprints of each incomplete defect. Another piece of information that is exchanged is the number of points in each complete and incomplete defect. This information is required for the categorization phase of the algorithm in Section 3.1.

Step 4: Inter-node aggregation. The process of growing the incomplete defects from different nodes is very similar to the process of growing incomplete defects from different chunks within a node. Therefore, we simply repeat the defect growing phase we had described as part of the Step 2 above.

After applying this step, we will have the set of defects which are formed after combining the defects from different nodes. This set, together with the defects which were complete on each node earlier, is the complete set of defects we are interested in.

Step 5: Defect re-distribution and categorization. Each node will work on its completed defects locally. However, defects whose growth was completed on the *master* node are divided up equally between processing nodes to achieve better load balance. This is done by assigning equal number of defects to each processor's space in the *reduction object*. If this communication operation was not performed then the execution times of our parallel implementation would not be scalable to the number of processing nodes, as demonstrated further in Section 3.3.2.

Matching is performed based on moment vector computed for a defect that is being categorized. This vector is compared to mean moment vectors for a number of classes available from the database. The number of classes represented in the database can have an effect on the application execution time, since defects that match require no further processing to be categorized, but *non-matching* defects need to have their "newly encountered" class added to the database. The defects that match are assigned to their respective classes then. Therefore, the processing nodes keep track of their *non-matching* defects for two distinct purposes:

1. to update the intermediate representation of the database used for matching the defects whose processing follows, and
2. to add the new defect classes to the catalog at the end of the categorization phase.

Step 6: Communicating non-matching defects to master node and updating the defect catalog. All *non-matching* defects are communicated to the *master* node as a part of the *reduction object*. All new defect classes are matched with each other using a *brute force* approach in order to make sure that only one representative per class is inserted into the catalog. After all the duplicates are removed, new permanent class names are assigned to the defect classes and the catalog is updated.

Step 7: Broadcast of class names and their update on the processing nodes. New class names are broadcasted back to the processing nodes to finalize the categorization of defects. The names are communicated as a part of the *reduction object*. The

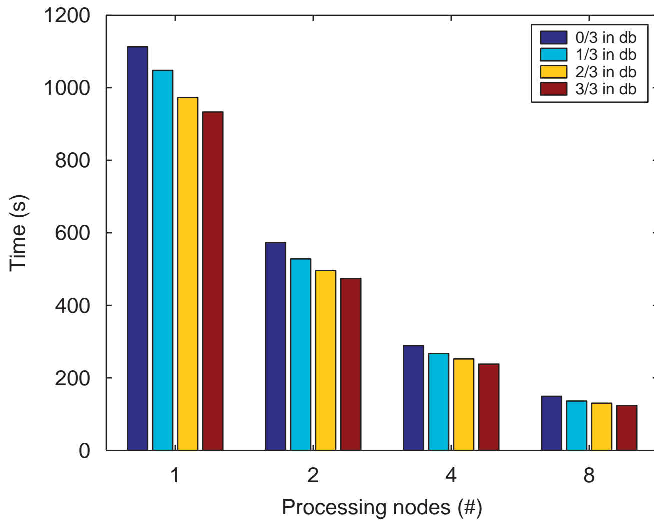

Fig. 2. Parallel performance on a 130 MB data set.

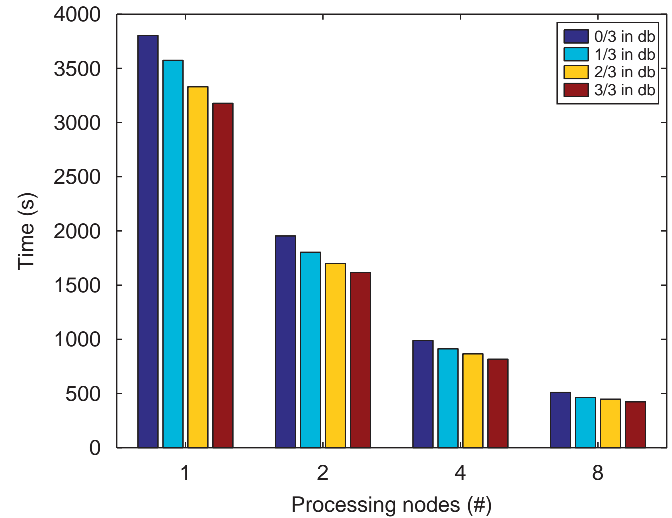

Fig. 3. Parallel performance on a 450 MB data set.

non-matching defects from each processing node are assigned new classes, in accordance to the names received in the broadcast.

3.3. Experimental results

In this section, we evaluate the performance of our parallel algorithm and implementation. We conducted a number of experiments with different data sets. One factor that impact the performance of the defect categorization phase is *database hit ratio*. Database hit ratio is defined as the percentage of defects that match the classes in the database or the catalog that is initially provided. We conducted experiments with different values of database hit ratio.

We had the following goals in our experiments: (1) studying the parallel scalability of our implementation, (2) investigating how database hit ratio effects the execution time of our parallel implementation, (3) evaluating the effect of our load balancing scheme, and (4) evaluating our approach to performing defect matching in parallel. Our experiments were conducted on a cluster of 700 MHz Pentium machines. The nodes in the cluster are connected through Myrinet LANai 7.0. The main memory on each node is 1 GB.

3.3.1. Evaluating scalability

This subsection reports experiments evaluating parallel scalability, with increasing data set sizes. We used three data sets, of sizes 130, 450 MB, and 1.8 GB, respectively. Each of these data sets were partitioned into eight chunks.

Fig. 2 presents the execution time for the 130 MB data set on 1, 2, 4, and 8 nodes of the cluster. We conducted experiments with four different values of database hit ratio, which were 0/3, 1/3, 2/3, and 3/3. On 2 nodes, the speedups range from 1.942 to 1.985. On 4 nodes, the speedups range from 3.851 to 3.925. Finally, on 8 nodes, the speedups range from 7.470 to 7.706. These results show that distributed memory parallelization works well, resulting in speedups which are very close to linear. Speedups are good even though the size of the data set is quite small, where we could have expected parallelization overheads to be significant. The variation in speedups with different values of hit ratios is quite small, though the speedups are a bit lower when this ratio is small. This is because both the amount of inter-processor communication and the sequential computation on the master node increase when there are fewer matches with the data set. However, the variation in speedups with different values of hit ratios is at most 3%, which demonstrates that our approach to dealing with defects that do not match the database is quite scalable.

Our second experiment evaluated parallel speedups on a larger data set (450 MB) with four different values of the hit ratio. Execution times on 1,2,4, and 8 nodes of the cluster are presented in Fig. 3. On 2 nodes, the speedups range from 1.946 to 1.982. On 4 nodes, the speedups range from 3.845 to 3.919. On 8 nodes, the speedups range from 7.457 to 7.703. These results are very similar to those from the experiment with the 130 MB data set, once again demonstrating that distributed memory parallelization is working well, parallelization overheads are small, and increasing the number of defects that do not match the database has only a small effect on the parallel performance.

Our next experiment was conducted to evaluate the parallel performance on a 1.8 GB data set. Fig. 4 demonstrates execution times on 1,2,4, and 8 nodes of a cluster, configured so that the hit ratio stays constant at 0/3. The figure presents:

- total execution time of detection and categorization phases combined (*total*),
- detection phase time (*detect*), and
- categorization phase time (*categorize*).

On 2 nodes, the speedups are 1.960 for *total*, 1.967 for *detect*, and 1.951 for *categorize*. On 4 nodes, the speedups are 3.841 for *total*, 3.890 for *detect*, and 3.783 for *categorize*. And, finally, on 8 nodes, the speedups are 7.425 for *total*, 7.493 for *detect*, and 7.345 for *categorize*. This experiment demonstrates that the efficiency of the parallel detection phase is slightly higher

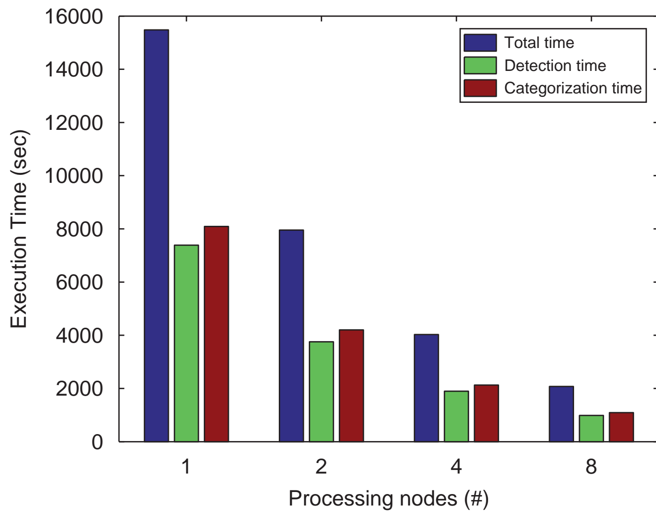

Fig. 4. Parallel performance on a 1.8 GB data set: defect detection and categorization stages (no defect matches the database).

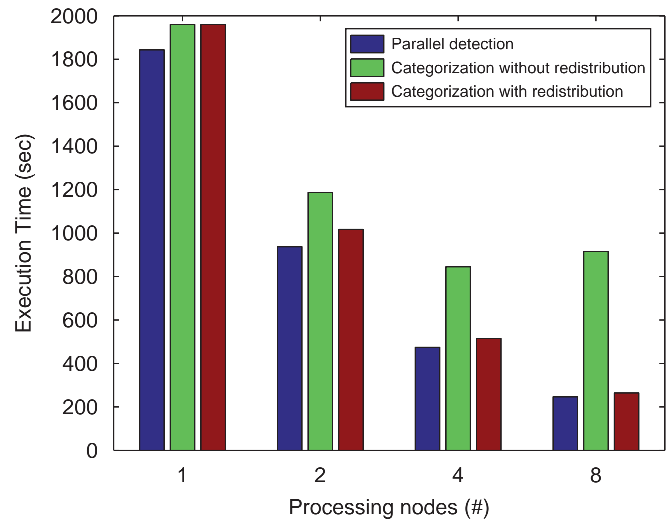

Fig. 6. Parallel categorization with and without load balancing (no defect matches the database).

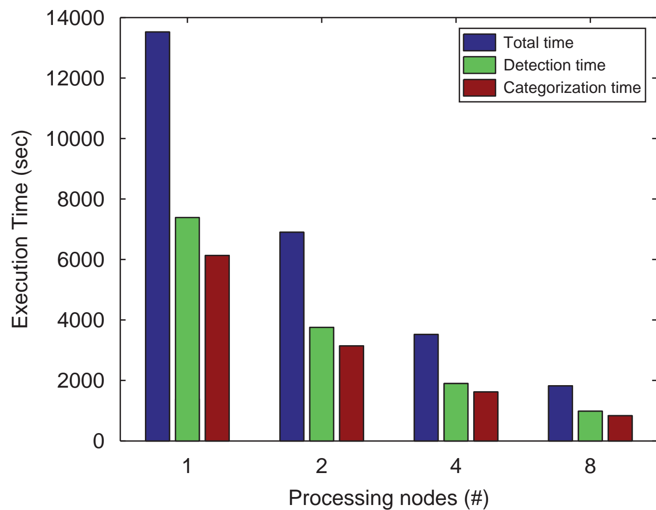

Fig. 5. Parallel performance on a 1.8 GB data set: defect detection and categorization stages (2/3 of the defects match the database).

than that of the parallel categorization phase, but the overall speedups for both are quite close to linear. Again, distributed memory parallelization is working well as the size of the data set has increased.

A similar experiment was conducted to evaluate the parallel performance on the same 1.8 GB data set, but this time with the database hit ratio being 2/3. Fig. 5 shows execution times on 1, 2, 4, and 8 nodes of the cluster, with the breakdown of execution times similar to Fig. 4. On 2 nodes, the speedups are 1.946 for *total*, 1.967 for *detect*, and 1.927 for *categorize*. On 4 nodes, the speedups are 3.845 for *total*, 3.890 for *detect*, and 3.804 for *categorize*. And, finally, on 8 nodes, the speedups are 7.457 for *total*, 7.493 for *detect*, and 7.424 for *categorize*.

Several observations can be made from this experiment. First, the speedups of the detection phase remained unchanged, since the hit ratio does not change the detection algorithm. Second, the categorization phase efficiency was lagging behind the detection phase efficiency, just as it was for the experiment described in Fig. 4. But since more defects matched the database in this experiment than in the previous one, less of the compute intensive matching had to be performed sequentially by the master node. This increased the parallel efficiency. Total execution time speedups, therefore, grew even closer to linear.

Overall, the four experiments we have presented in this subsection show that the processing time is mostly proportional to the size of the data set, and that the parallel efficiency is not greatly effected by the increased size of the problem.

3.3.2. Evaluating effects of the load balancing scheme

One of the significant aspects of our implementation was the scheme used for load balancing for the defect categorization phase. In this subsection, we evaluate the impact of this scheme on the parallel performance.

The first experiment was conducted to compare parallel detection time with parallel categorization time with and without the *redistribution*. In the implementation without the redistribution, defects that span more than 1 node are categorized sequentially on the master node. Fig. 6 shows results from the experiment performed using the 450 MB data set, with the hit ratio being 0/3. On 1, 2, 4, and 8 nodes, we present execution times for the parallel detection, categorization without redistribution, and categorization with redistribution. The time for the parallel detection phase is presented as a baseline, because, as we saw in Section 3.3.1, the detection phase achieves near linear speedups.

The categorization version without redistribution gets significantly slower as the number of nodes increase. The speedups of categorization without redistribution were 1.65 on 2 nodes, 2.32 on 4 nodes, and 2.14 on 8 nodes. In comparison, the speedups of both the parallel detection and categorization with redistribution are almost linear. For the entire application, the speedup on 8 nodes will be only 3.3 if we used categorization

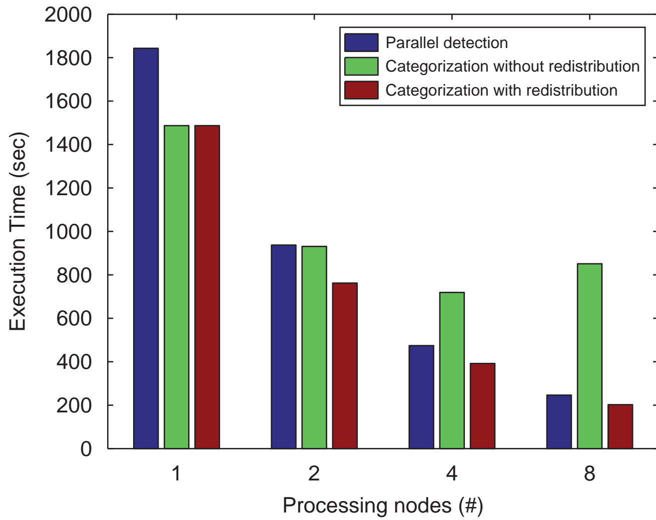

Fig. 7. Parallel categorization with and without load balancing (2/3 defects match the database).

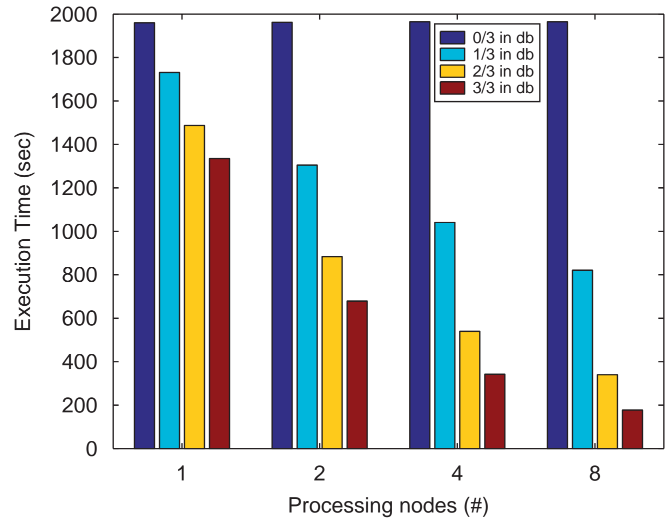

Fig. 8. Sequential categorization of non-matching defects: 450 MB data set.

without redistribution. This shows that the redistribution performed in our load balancing scheme is critical for parallel scalability.

In Fig. 7, we show results from a similar experiment, the only difference being the hit ratio, which is now 2/3. The speedups for defect detection and categorization with redistribution were once again near linear. For categorization without redistribution, the speedups were 1.60 on 2 nodes, 2.07 on 4 nodes, and 1.75 on 8 nodes. These results, again, demonstrate that redistribution achieves acceptable load balance, whereas without such redistribution, parallel efficiency is low. Also, as we partition our lattice across more nodes, the number of defects that span more than 1 node increases. This is why sequentializing the categorization of such defects gives us worse performance on 8 nodes than on 2 nodes.

3.3.3. *Evaluating parallel matching approach*

Another important aspect of our implementation was how we parallelize categorization of non-matching defects. In this subsection, we evaluate our approach and compare it to the *naive* approach, in which we can send all non-matching defects to the master node and categorize them sequentially.

Fig. 8 summarizes the parallel execution times of the naive approach. We use a 450 MB data set with the hit ratio varied between 0/3 and 3/3. The performance of naive version depends heavily on the hit ratio. When the hit ratio is 0/3, the execution times for the categorization phase do not scale at all. When the hit ratio is 3/3, the speedups are near linear. This is because categorization is sequentialized when no defects match the database. In comparison, when all the defects match the database, the naive version is not really different from the optimized version. The results when the hit ratio is 1/3 or 2/3 are consistent with our expectations, i.e., the naive version only achieves modest speedups.

The parallel performance of our optimized version is shown in Fig. 9. Unlike the naive version, our approach for

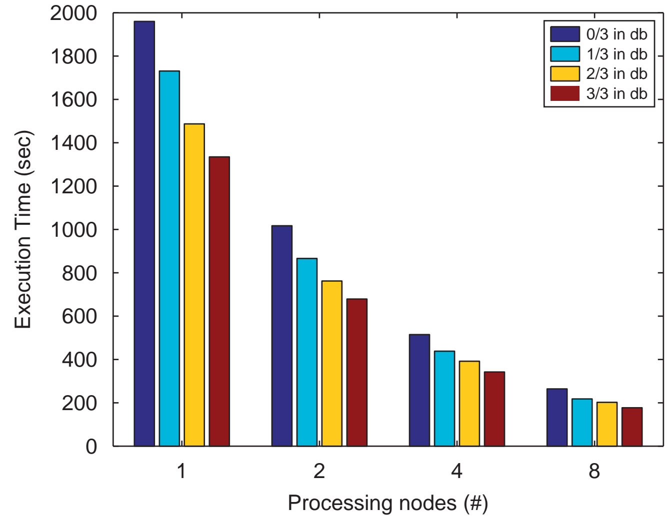

Fig. 9. Parallel categorization of non-matching defects: 450 MB data set.

parallelizing this step achieved almost linear speedups for all four values of hit ratio.

4. FREERIDE-G: from clusters to grid

FREERIDE-G is an extension of FREERIDE which targets processing of data stored in remote repositories.

4.1. *System design*

This subsection describes the overall design of the FREERIDE-G middleware. The basic functionality of the system is to automate retrieval of data from remote repositories and coordinate parallel analysis of such data using end-user's computing resources, provided an inter-connection exists between the repository disk and the end-user's computing nodes. This system expects data to be stored in chunks, whose size is manageable for the repository nodes.

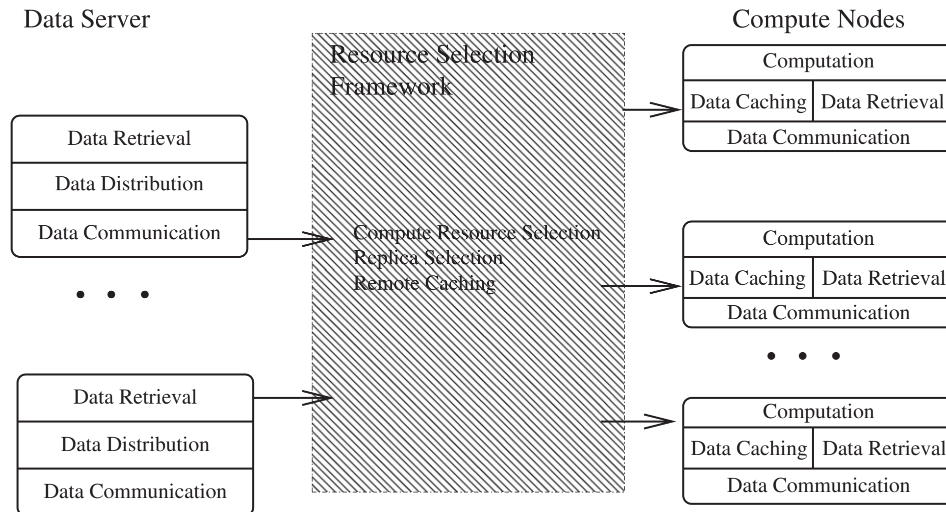

Fig. 10. FREERIDE-G system architecture.

This middleware is modeled as a *client–server* system. Fig. 10 shows the three major components, including the *data server*, the *compute node client*, and a *resource selection framework*. As we stated earlier, the resource selection framework is part of our ongoing work on FREERIDE-G, and is beyond the scope of this paper.

The data server runs on every on-line data repository node in order to automate data delivery to the end-users processing node(s). More specifically, it has three roles:

1. *Data retrieval*: Data chunks are read in from repository disk.
2. *Data distribution*: Each data chunk is assigned a destination—a specific processing node in the end-user's system.
3. *Data communication*: After destination assignment is made in the previous step, each data chunk is sent to the appropriate processing node.

A compute server runs on every end-user processing node in order to receive the data from the on-line repository and perform application specific analysis of it. This component has four roles:

1. *Data communication*: Data chunks are delivered from a corresponding data server node.
2. *Data retrieval*: If caching was performed on the initial iteration, each subsequent pass retrieves data chunks from local disk, instead of receiving it via network.
3. *Computation*: Application specific data processing is performed on each chunk.
4. *Data caching*: If multiple passes over the data chunks will be required, the chunks are saved to a local disk.

The current implementation of the system is configurable to accommodate N data server nodes and M user processing nodes between which the data has to be divided, as long as $M \geqslant N$. The reason for not considering cases where $M < N$ is that our target applications involve significant amount of computing, and cannot effectively process data that is retrieved from a larger number of nodes.

The configuration illustrated in Fig. 10 presents a setup with $N = 2$ data servers and $M = 3$ compute nodes. Active data repository (ADR) [6,7] was used to automate the data retrieval parts of both components.

4.2. System implementation issues

This section describes a number of implementation issues in the FREERIDE-G middleware system. The main issues are: managing and communicating remote data, load distribution, parallel processing on compute nodes, and caching of remote data.

4.2.1. Managing and communicating remote data

As we stated in the previous section, data are organized as chunks on remote repositories, using an existing ADR middleware. The processing of data is organized in *phases*. In each phase, a generalized reduction is performed on the computing nodes. Because of the property of reductions, the order of retrieving, communicating, and processing data elements does not impact the correctness.

At the beginning of each phase, the compute nodes forward the information on the subset of the data to be processed to data server. The data server determines the chunks of the data that need to be retrieved, as well as a schedule for retrieving these on each data server node.

Initially, let us suppose that the number of data server nodes equals the number of compute nodes. In such a scenario, each data server node forwards all the chunks it retrieves to a single compute node. The support for declustering of chunks in ADR helps maintain a good balance, even with such a simple scheme. The corresponding data server and compute nodes coordinate when the next chunk should be communicated, and also the size of the buffer that needs to be allocated on the compute node. In our current implementation, stream socket mechanism was used for all such communication.

4.2.2. Load distribution

Data mining and scientific processing applications are often compute-intensive. In such cases, they can benefit from a configuration where the number of compute nodes is larger than the number of data server nodes. However, in such cases, careful load distribution must be performed.

We again use a simple mechanism. Each data server node now communicates its chunks to M compute nodes. The value M is the smallest value which will still enable load balance on each compute node. A hash function (*mod*) based on a unique chunk id is used to distribute the retrieved chunks among the M compute nodes a data server node is communicating with.

4.2.3. Caching

If an iterative mining application needs to take more than a single pass over the data, reading the data from the remote location on every iteration is redundant. For such applications, data chunks belonging to a certain compute node can be saved onto the local disk, provided sufficient space. Such *caching* is performed during the initial iteration, after each data chunk is communicated to its compute node by the data server and the first pass of application specific processing has been completed.

Each chunk is written out to the compute node's disk in a separate file, whose name is uniquely defined by the chunk id. These filenames are also indexed by the chunk ids, speeding up retrieval for the subsequent iterations. The benefit of such caching scheme is evident: for an application requiring P passes over the data, the last $P - 1$ iterations will have the data available locally on the compute node. Since each round out data communication from the server would have to perform retrieval in order to send the data, the total number of retrievals does not change. Instead, for iterations subsequent to the initial one, data retrieval is performed on the compute node.

5. Experimental results from FREERIDE-G

In this section, we evaluate the performance of the FREERIDE-G middleware. We use the five data analysis applications described in Section 5.1. Several different data sets, of varying sizes, were used for each of these. We had the following goals in our experiments:

1. Studying parallel scalability of applications developed using FREERIDE-G. Here, we focused on configurations where the numbers of compute and data repository nodes are always equal.
2. Investigating how the computing can be scaled, i.e., performance improvements from increasing the number of compute nodes independent of the number of data server nodes.
3. Evaluating the benefits of performing caching in applications that require multiple passes over data.

For efficient and distributed processing of data sets available in a remote data repository, we need high bandwidth networks and a certain level of quality of service support. Recent trends are clearly pointing in this direction. However, for our study, we did not have access to a wide-area network that gave high bandwidth and allowed repeatable experiments. Therefore, all our experiments were conducted within a single cluster. The cluster used for our experiment comprised 700 MHz Pentium machines connected through Myrinet LANai 7.0. In experiments involving caching, the communication bandwidth was simulated to be 500 KB/s and 1 MB/s.

5.1. Applications

In this section we describe the applications that we have used to carry out the experimental evaluation of our middleware. We have focused on three traditional data mining techniques: k-means clustering [20], EM clustering [14], k-nearest neighbor search [19], as well as two scientific feature mining algorithms: vortex analysis [32] and molecular defect detection [34]. As molecular defect detection was described earlier in this paper, we only present parallelization details of the first four applications.

5.1.1. k-Means clustering

The first data mining algorithm we describe is the k-means clustering technique [20], which is one of the most popular and widely studied data mining algorithms. This method considers data instances represented by points in a high-dimensional space. Proximity within this space is used as criterion for classifying the points into clusters.

Three steps in the sequential version of this algorithm are as follows:

1. start with k given centers for clusters;
2. scan the data instances. For each data instance (point), find the center closest to it, assign this point to a corresponding cluster, and then move the center of the cluster closer to this point; and
3. repeat this process until the assignment of the points to cluster does not change.

This method can be parallelized as follows. The data instances are partitioned among the nodes. Each node processes the data instances it owns. Instead of moving the center of the cluster immediately after the data instance is assigned to the cluster, the *local sum* of movements of each center due to all points owned on that node is computed. A *global reduction* is performed on these local sums to determine the centers of clusters for the next iteration.

5.1.2. Expectation maximization clustering

The second data mining algorithm we have used is the expectation maximization (EM) clustering algorithm [14], which is one of the most popular clustering algorithms. EM is a distance-based algorithm that assumes the data set can be modeled as a linear combination of multi-variate normal distributions. The goal of the EM algorithm is to use a sequence of expectation and maximization steps to estimate the means C, the covariances R, and the mixture weights W of a Gaussian probability function. The algorithm works by successively improving the solution found so far. The algorithm stops when the quality of the current solution becomes stable, which is measured by a monotonically increasing statistical quantity called *loglikelihood*.

This algorithm can be parallelized in the following manner. The input data instances (the array *Y*) are distributed between the nodes. The arrays *C*, *R*, and *W*, whose initial values are provided by the user, are replicated on all nodes. The *E* step is carried out on each node, using data instances local to it. *Global combination* involved in the *E* step consists of the information necessary to compute the means and mixture weight arrays being aggregated by the master node, and then being re-broadcasted. Next, the *M* step is performed locally on each node's data instances. Information necessary to compute covariance is then updated during the *M* step, through an aggregation step followed by a re-broadcast.

At the end of any iteration, each node has an updated value for *C*, *R*, *W* and *llh*, and the decision to execute or abort another iteration is made locally.

These parallelization steps can be expressed easily using the FREERIDE-G API described earlier in this paper [16].

5.1.3. k-Nearest neighbor search

k-Nearest neighbor classifier is based on learning by analogy [19]. The training samples are described by an *n*-dimensional numeric space. Given an unknown sample, the k-nearest neighbor classifier searches the pattern space for k training samples that are closest, using the euclidean distance as measure of proximity, to the unknown sample.

Again, this technique can be parallelized as follows. The training samples are distributed among the nodes. Given an unknown sample, each node processes the training samples it owns to calculate the k-nearest neighbors *locally*. After this local phase, a global reduction computes the overall k-nearest neighbors from the k-nearest neighbor on each node.

5.1.4. Vortex detection algorithm

Vortex detection is the first of the two scientific data processing applications we have used. Particularly, we have parallelized a feature mining based algorithm developed by Machiraju et al. A more detailed overview of the algorithm is available in a recent publication [37]. The key to the approach is extracting and using *volumetric regions* to represent features in a CFD simulation output.

This approach identifies individual points (*detection step*) as belonging to a feature (*classification step*). It then aggregates them into regions. The points are obtained from a tour of the discrete domain and can be in many cases the vertices of a physical grid. The sensor used in the detection phase and the criteria used in the classification phase are physically based point-wise characteristics of the feature. For vortices, the detection step consists of computing the eigenvalues of the velocity gradient tensor at each field point. The classification step consists of checking for complex eigenvalues and assigning a swirl value if they exist. The aggregation step then defines the region of interest (ROI) containing the vortex. Regions insignificant in size are then eliminated, and the remaining regions are sorted based on a certain parameter (like size or swirl).

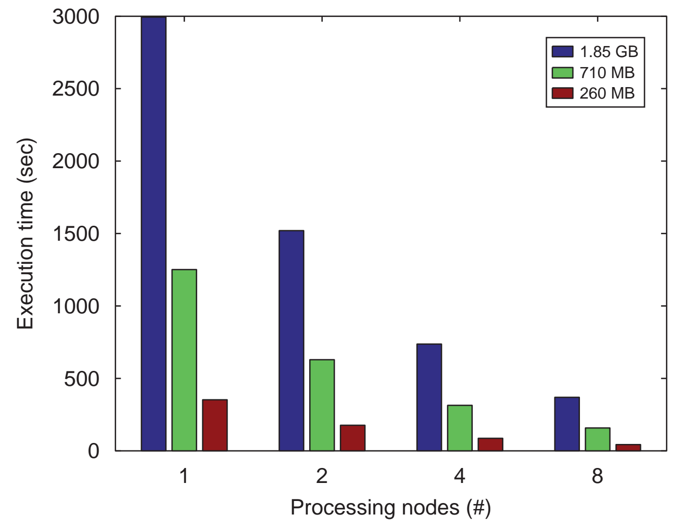

Fig. 11. Vortex detection application parallel performance on 1.85 GB, 710, and 260 MB data sets.

Parallelizing this application requires the following steps [17]. First, when data are partitioned between nodes, an overlap area between data from neighboring partitions is created, in order to avoid communication in the detection phase. Detection, classification and aggregation are first performed locally on each node, followed by *global combination* that joins parts of a vortex belonging to different nodes. Denoising and sorting of vortices is performed after the final aggregation has been completed.

5.2. Evaluating overall system scalability

The number of compute nodes used for these experiments was always equal to the number of data repository nodes. In this situation pair-wise correspondence between data and compute nodes can be established, and no distribution of data to multiple compute nodes is required from the data server. All scalability experiments were conducted on up to 16 nodes (eight data and compute node pairs).

Vortex detection was evaluated with three data sets, with size of 260, 710 MB, and 1.85 GB, respectively. Fig. 11 presents the execution times from these three data sets on 1, 2, 4, and 8 pairs of nodes. On 2 pairs of nodes, the speedups are 1.99 for the 260 MB data set, 1.98 for 710 MB data set, and 1.97 for the 1.8 GB data set. This demonstrates that distributed memory parallelization is working very well, resulting in nearly perfect speedups. Speedups are good even for the smallest data set, where execution time is expected to be mostly dominated by the parallelization overhead. Also, since data communication overhead is kept relatively low, communication time scales as well with data size as data retrieval and analysis times.

On 4 pairs of nodes, the speedups are 3.99 for the 260 MB data set, 3.98 for 710 MB data set, and 3.96 for the 1.8 GB data set. On 8 pairs of nodes, the speedups are 7.95 for the 260 MB data set, 7.92 for 710 MB data set, and 7.90 for the 1.8 GB data set.

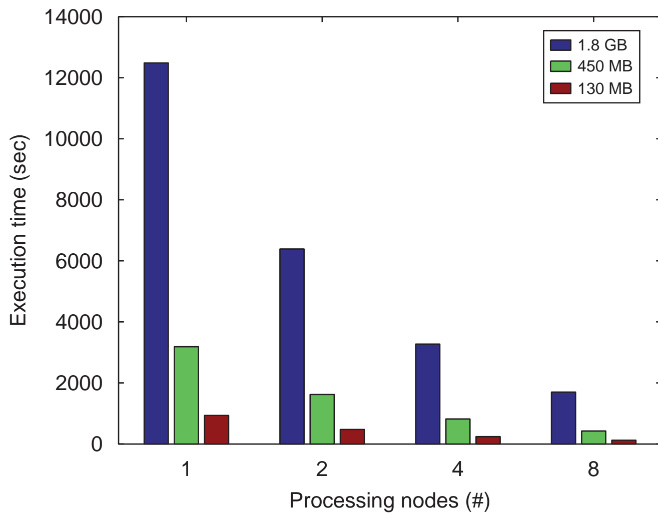

Fig. 12. Defect detection application parallel performance on 1.8 GB, 450, and 130 MB data sets.

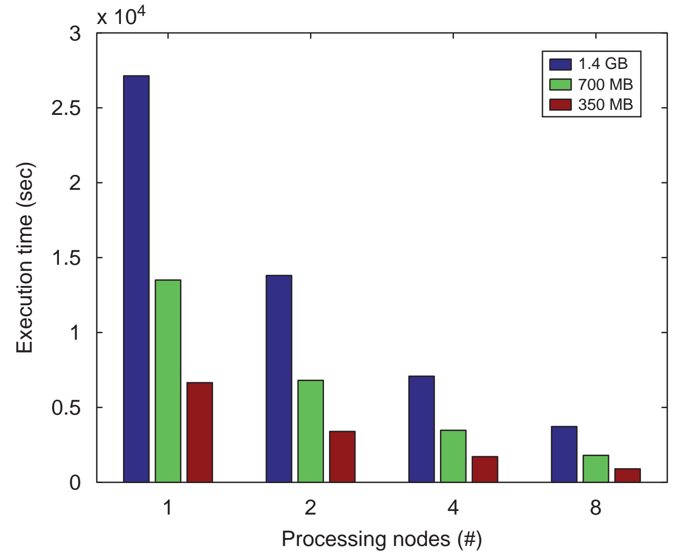

Fig. 13. Expectation maximization clustering parallel performance on 1.4 GB, 700, and 350 MB data sets.

Fig. 12 presents parallel execution times for the molecular defect detection algorithm. This application was evaluated on three data sets of sizes 130, 450 MB, and 1.8 GB. On 2 pairs of nodes, the speedups in execution time were 1.97 for the 130 MB data set, 1.97 for the 450 MB data set and 1.96 for the 1.8 GB data set. Again, near perfect speedups demonstrate good parallelization efficiency.

On 4 pairs of nodes, the speedups were 3.92 for the 130 MB data set, 3.89 for the 450 MB data set and 3.82 for the 1.8 GB data set. The drop-off in speedups here demonstrates that the overhead associated with communication between compute nodes that is required for defect detection is not as small as that for vortex detection. But, with parallel efficiency somewhat limited by the application itself, the speedups are still very good. On 8 pairs of nodes, the speedups are 7.52 for the 130 MB data set, 7.50 for the 450 MB data set and 7.34 for the 1.8 GB data set.

Figs. 13–15 present execution times from the additional scalability experiments that were conducted. EM clustering, k-means clustering, and k-nearest neighbor search were evaluated on three data sets of size 350, 700 MB, and 1.4 GB.

On 8 pairs of nodes, parallel EM achieved speedups of 7.56 for 350 MB data set, 7.49 for 700 MB data set and 7.30 for 1.4 GB data set. In the same configuration, parallel k-means achieved speedups of 7.25 for 350 MB data set, 7.21 for 700 MB data set and 7.10 for 1.4 GB data set. Parallel k-nearest neighbor search, executed on 8 pairs of nodes, achieved speedups of 7.26 for 350 MB data set, 7.15 for 700 MB data set and 6.98 for 1.4 GB data set.

Results were once again consistent with those of the previous two experiments. Parallel efficiency observed was high, although in some cases limited by the application. Data retrieval, communication and processing all demonstrated good scalability with respect to increasing both the problem size and the number of compute nodes.

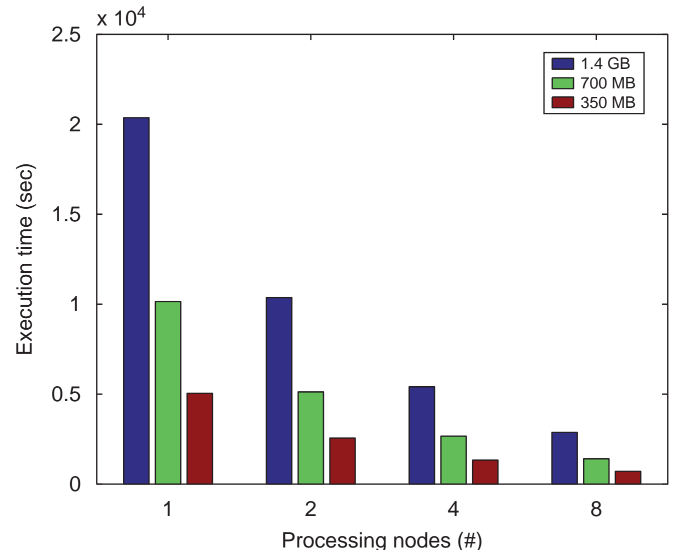

Fig. 14. K-means clustering parallel performance on 1.4 GB, 700, and 350 MB data sets.

5.3. Evaluating scalability of compute nodes

In processing data from remote repositories, the number of available nodes for processing may be larger than the number of nodes on which data is hosted. As we described earlier, our middleware can support processing in such configurations. In this subsection, we evaluate the performance of applications in such cases.

We used three of the five applications, i.e., defect detection, vortex detection and k-nearest neighbor search, for these experiments. Unlike the other two applications (k-means and EM clustering), each of these three applications only take a single pass (of retrieval and communication) over the data. So, any

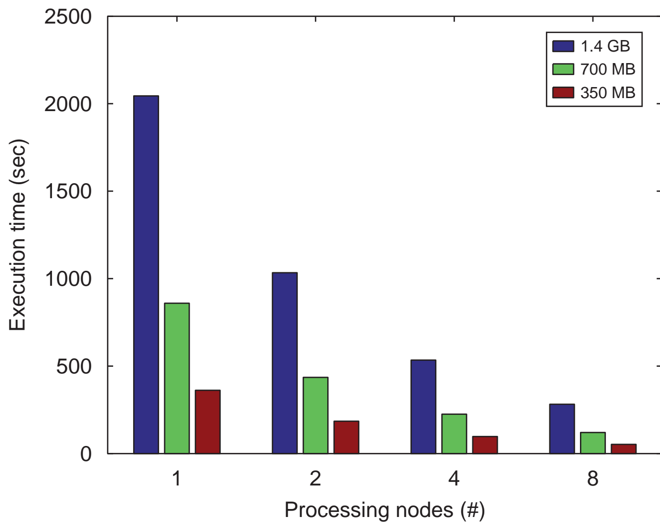

Fig. 15. k-nearest neighbor search parallel performance on 1.4 GB, 700, and 350 MB data sets.

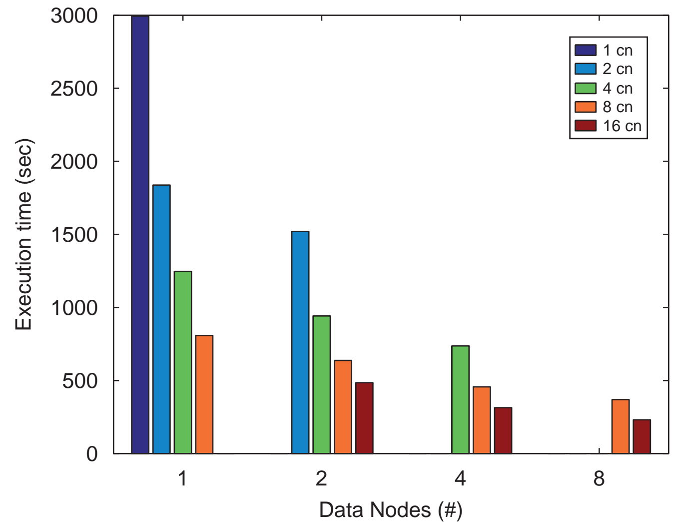

Fig. 17. Vortex detection parallel performance as the number of compute nodes is scaled (1.85 GB data set).

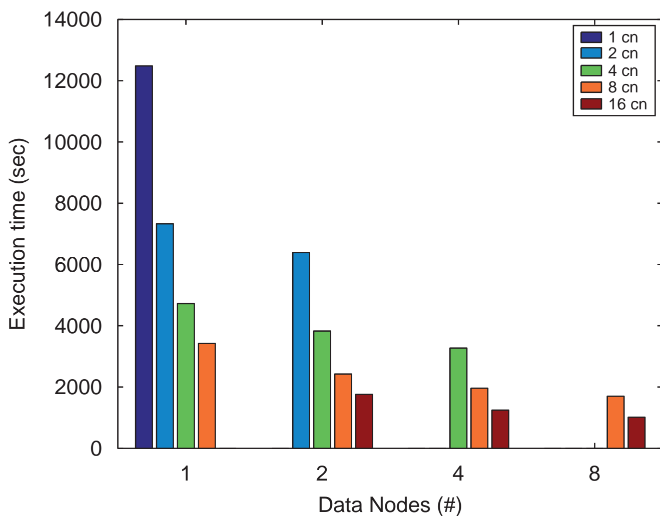

Fig. 16. Defect detection parallel performance as the number of compute nodes is scaled (1.8 GB data set).

change in performance achieved by the middleware would be due to each data node distributing processing work to multiple compute nodes, and not due to caching.

Among the data sets used in the experiments in the previous subsection, we report results from only the largest ones. The number of data nodes was varied up to 8 and the number of compute nodes was varied up to 16 for each experiment. While both numbers were restricted to be powers of two to achieve perfect load balance, nothing in the middleware implementation requires such restriction.

Fig. 16 presents parallel defect detection execution times on a 1.8 GB data set, as the number of both data nodes and compute nodes was varied. Using a single compute node, the speedups achieved were 1.70 for 2 compute nodes, 2.64 for 4, and 3.65 for 8. The speedups are sub-linear because only the data processing work is being parallelized, with data retrieval and communication tasks remaining sequential. However, these experiments do show that in cases where additional compute nodes are available, our middleware can use them to obtain further speedups, even if these speedups are sub-linear.

Using two data nodes, the additional speedups achieved were 1.67 for 4 compute nodes, 2.63 for 8, and 3.63 for 16. With four data nodes, the speedups were 1.67 for 8 compute nodes, and 2.62 for 16. And, finally, using eight data and 16 compute nodes, the speedup was 1.67. These results demonstrate that a very decent speedup can be achieved by using twice as many compute nodes as data nodes, but as the number of compute nodes keeps increasing, a drop off in parallel efficiency is to be expected.

Fig. 17 presents parallel vortex detection execution times on a 1.85 GB data set. Again, the number of both data and compute nodes is varied. Using a single data node, the speedups achieved were 1.63 for 2 compute nodes, 2.40 for 4, and 3.61 for 8. Again, speedups are sub-linear because only a fraction of execution time has been parallelized. In fact, a larger fraction of time is spent on data retrieval in the vortex detection application, resulting in slightly lower speedups. Using two data nodes, the additional speedups are 1.61 for 4 compute nodes, 2.39 for 8, and 3.14 for 16. The lower speedup of the last configuration is attributed to parallelization overhead starting to dominate over execution time. With four data nodes, the speedups achieved were 1.61 for 8 data nodes, and 2.35 for 16. And, finally, using eight data and 16 compute nodes, the speedup was 1.60. These results are consistent with the defect detection experiment, only indicating a slightly higher tendency for vortex detection to be "I/O bound."

Fig. 18 presents parallel execution times for k-nearest neighbor search evaluated on the 1.4 GB data set. Once again, the number of data and compute nodes is varied. Using a single data node, the speedups achieved were 1.48 on 2 compute nodes, 1.98 on 4, and 2.38 on 8. This indicates that the fraction of

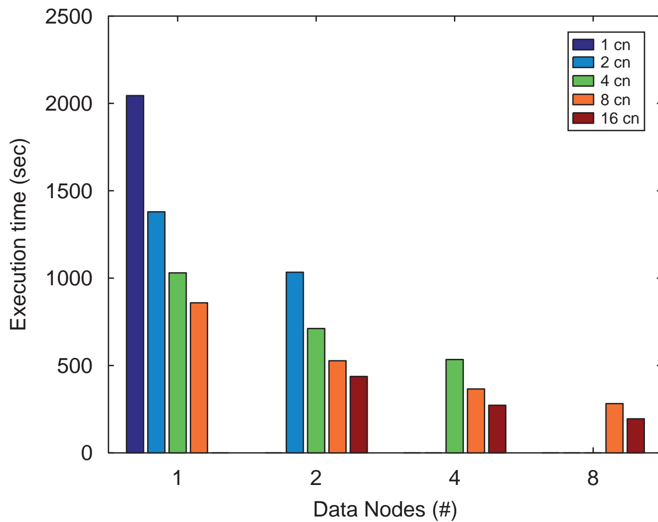

Fig. 18. k-nearest neighbor search parallel performance as the number of compute nodes is scaled (1.4 GB data set).

time spent on data retrieval is even higher for this application. Again, as a larger fraction of execution time remains sequentialized, the speedup decreases. With two data nodes, the additional speedups achieved are 1.45 on 4 compute nodes, 1.96 on 8, and 2.36 on 16. These results are consistent with previous experiments with both this application and other applications. Using four data nodes, the speedups achieved are 1.46 on 8 compute nodes, and 1.96 for 16. Finally, using eight data and 16 compute nodes, the speedup was 1.44.

Overall, the results indicate that scaling up the number of compute nodes beyond the number of data nodes results in a more modest speedup than scaling both compute and data nodes. However, these results do show that additional computing nodes can be used to decrease processing rates.

5.4. Evaluating effects of caching

When a data processing application involves multiple passes over data, FREERIDE-G supports the ability to cache remote data. This subsection describes experiments evaluating the benefits of such caching. We use the two multi-pass applications from our set of applications, which are k-means and EM clustering. As the results from these two applications were very similar, we are only presenting results from EM in this subsection. We executed this application for five iterations, and used simulated cluster inter-connection bandwidth of 500 KB/s and 1 MB/s.

As in Section 5.2 three data sets of size 350, 700 MB, and 1.4 GB, respectively, were used. Two versions were created: `Cache` version utilizes a caching framework, as described in Section 4.2.3, and the `No cache` version, which does not save the data locally during the initial iteration, and, therefore, requires that the server node communicates it again to the compute node during each subsequent iteration.

Fig. 19 demonstrates a comparison of parallel execution times of the `cache` and `no cache` versions of the EM

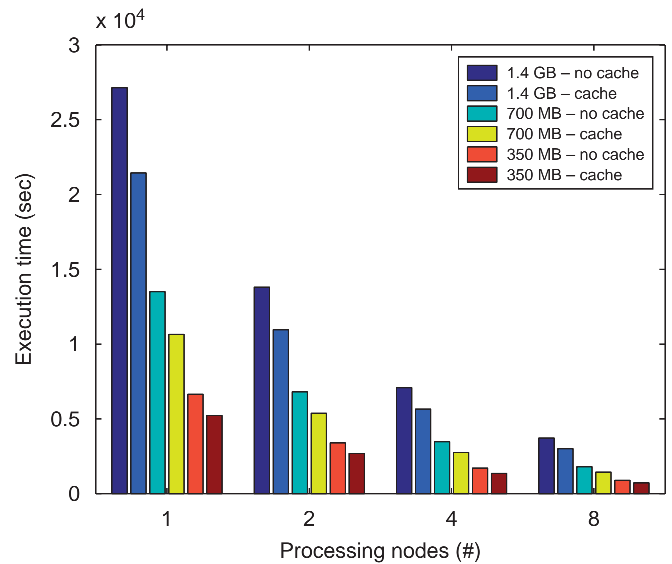

Fig. 19. Comparing EM performance with and without caching on 350, 700 MB, and 1.4 GB data sets (1 MB/s bandwidth).

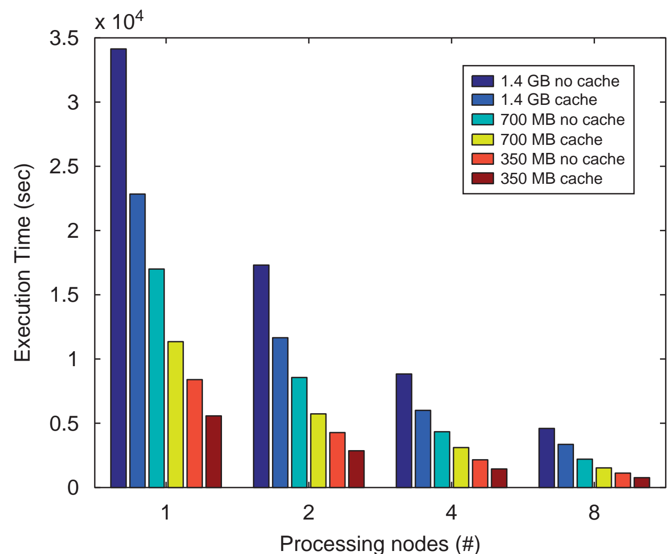

Fig. 20. Comparing EM performance with and without caching on 350, 700 MB, and 1.4 GB data sets (500 KB/s bandwidth).

clustering application, with 1 MB/s bandwidth. In all 1-to-1 parallel configurations across all three data sets, the decrease in execution time due to caching is around 1.27. This demonstrates that there is a significant benefit to caching the data locally. In fact, when the breakdown of the execution times were considered, data communication time for the `cache` version was about 20% of the same time for the `no cache` version. Such results were to be expected, since `cache` communicates data only once, whereas `no cache` communicates it five times, once per iteration.

Finally, Fig. 20 illustrates the caching benefits for the EM application, but with communication bandwidth of 500 KB/s.

Parallel EM in this setup demonstrates a speedup of around 1.51 in all 1-to-1 parallel configurations, across three data sets.

Overall, caching experiments presented demonstrate that the relative benefit achieved from our caching framework is relatively independent of the size of the problem or the parallel configuration. Instead, communication bandwidth available and the ratio of communication time to compute time determine the factor of improvement in execution times.

6. Related work

One effort somewhat similar to our cluster middleware effort is from Becuzzi et al. [2]. They use a structured parallel programming environment PQE2000/SkIE for developing parallel implementation of data mining algorithms. Darlington et al. [13] have also used structured parallel programming for developing data mining algorithms. Our work is distinct in at least two important ways. First, they only target distributed memory parallelism (while they report results on an SMP machine, it is using MPI). Second, I/O is handled explicitly by the programmers in their approach. Goil and Choudhary have developed PARSIMONY, which is an infrastructure for analysis of multi-dimensional data sets, including OLAP and data mining [18]. PARSIMONY does not offer high-level interfaces, starting from which parallelization and I/O optimization may be achieved.

Several groups have been developing support for grid-based data mining. One effort in this area is from Cannataro et al. [4,5]. They present a structured Knowledge Grid toolset for developing distributed data mining applications through workflow composition. Brezanny et al. [3,23,31] have also developed a GridMiner toolkit for creating, registering and composing data mining services into complex distributed and parallel workflows. Ghanem et al. [12,15] have developed Discovery Net, an application layer for providing grid-based services allowing creation, deployment and management of complex data mining workflows. The goal of DataMiningGrid, carried out by Stankovski et al. [36], is to serve as a framework for distributed knowledge discovery on the grid.

There are significant differences between these efforts and our work. These systems do not offer a high-level interface for easing parallelization and abstracting remote data extraction and transfer. We believe that FREERIDE-G is able to reduce the time required for developing applications that perform remote data analysis. On the other hand, our system is not yet integrated with Grid standards and services.

Jacob et al. have created GRIST [21], a grid middleware for astronomy related mining. This effort, however, is very domain specific, unlike FREERIDE-G, which has been used for a variety of data mining and scientific analysis algorithms.

Much work has been done on parallelization of classification algorithms [30,33,35,38]. The algorithm for defect categorization we parallelize is very different than the algorithms considered in these efforts, and therefore, the issues in parallelization are quite different.

Several researchers have parallelized feature extraction algorithms, especially, in the context of computer vision. This includes the work from Chung and Prasanna [11] and Chen and Silver [9]. Our work is distinct in two important ways. First, we also parallelize the defect categorization phase. Second, we have shown how a cluster middleware could be used for both parallelization and scaling on disk-resident data sets.

7. Conclusions

This paper has given an overview of two middleware systems that have been developed over the last 6 years to address the challenges involved in developing parallel and distributed implementations of data mining algorithms. FREERIDE focuses on data mining in a cluster environment. FREERIDE-G supports a high-level interface for developing data mining and scientific data processing applications that involve data stored in remote repositories. The added functionality in FREERIDE-G aims at abstracting the details of remote data retrieval, movements, and caching from application developers.

This paper has presented some of the application development efforts and experimental results we have obtained from these two systems. Specifically, we have described our experience from developing a molecular defect detection application on FREERIDE. We have also presented initial performance evaluation of FREERIDE-G using three data mining algorithms and two scientific data processing applications.

References

[1] R. Agrawal, J. Shafer, Parallel mining of association rules, IEEE Trans. Knowl. Data Eng. 8 (6) (1996) 962–969.

[2] P. Becuzzi, M. Coppola, M. Vanneschi, Mining of association rules in very large databases: a structured parallel approach, in: Proceedings of Europar-99, Lecture Notes in Computer Science (LNCS), vol. 1685, Springer, Berlin, August 1999, pp. 1441–1450.

[3] P. Brezany, J. Hofer, A. Tjoa, A. Wohrer, Gridminer: an infrastructure for data mining on computational grids, in: Proceedings of Australian Partnership for Advanced Computing Conference (APAC), Gold Coast, Australia, October 2003.

[4] M. Cannataro, A. Congiusta, A. Pugliese, D. Talia, P. Trunfio, Distributed data mining on grids: services, tools, and applications, IEEE Trans. Systems Man Cybernet. Part B 34 (6) (2004) 2451–2465.

[5] M. Cannataro, D. Talia, KNOWLEDGE GRID: an architecture for distributed knowledge discovery, Comm. ACM 46 (1) (2003) 89–93.

[6] C. Chang, A. Acharya, A. Sussman, J. Saltz, T2: a customizable parallel database for multi-dimensional data, ACM SIGMOD Record 27 (1) (1998) 58–66.

[7] C. Chang, R. Ferreira, A. Acharya, A. Sussman, J. Saltz, Infrastructure for building parallel database systems for multidimensional data, in: Proceedings of the Second Merged IPPS/SPDP (13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing). IEEE Computer Society Press, Silva Spring, HD, April 1999.

[8] P. Cheeseman, J. Stutz, Bayesian classification (autoclass): theory and practice, in: Advanced in Knowledge Discovery and Data Mining, pp. 61–83. AAAI Press, MIT Press, Cambridge, Ma, 1996.

[9] J. Chen, D. Silver, Distributed feature extraction and tracking, in: Proceedings of SPIE Conference on Vizualization and Data Analysis, 2002.

[10] A. Chervenak, I. Foster, C. Kesselman, C. Salisbusy, S. Tuecke, The data grid: towards an architecture for the distributed management and analysis of large scientific data sets, J. Network Comput. Appl. 23 (3) (2001) 187–200.

[11] Y. Chung, V. Prasanna, Parallelizing image feature extraction on coarse-grain machines, IEEE Trans. Pattern Anal. Mach. Intell. (PAMI) (12) (1998) 1389–1394.
[12] V. Curcin, M. Ghanem, Y. Guo, M. Kohler, A. Rowe, J. Syed, P. Wendel, Grid knowledge discovery processes and an architecture for their composition, in: The Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Alberta, Canada, July 2002.
[13] J. Darlington, M.M. Ghanem, Y. Guo, H.W. To, Performance models for co-ordinating parallel data classification, in: Proceedings of the Seventh International Parallel Computing Workshop (PCW-97), Canberra, Australia, 1997.
[14] A. Dempster, N. Laird, D. Rubin, Maximum likelihood estimation from incomplete data via the EM algorithm, J. Roy. Statist. Soc. 39 (1) (1977) 1–38.
[15] M. Ghanem, Y. Guo, A. Rowe, P. Wendel, Grid-based knowledge discovery services for high throughput informatics, in: The Eleventh IEEE International Symposium on High Performance Distributed Computing, Edinburgh, Scotland, 2002.
[16] L. Glimcher, G. Agrawal, Parallelizing EM clustering algorithm on a cluster of SMPs, in: Proceedings of Europar, 2004.
[17] L. Glimcher, X. Zhang, G. Agrawal, Scaling and Parallelizing a Scientific Feature Mining Application Using a Cluster Middleware, in: Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS), 2004.
[18] S. Goil, A. Choudhary, PARSIMONY: an infrastructure for parallel multidimensional analysis and data mining, J. Parallel Distributed Comput. 61 (3) (2001) 285–321.
[19] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann Publishers, Los Altos, CA, 2000.
[20] J.A. Hartigan, M.A. Wong, A k-means clustering algorithm, Appl. Statistics (28) (1979) 100–108.
[21] J.C. Jacob, R. Williams, J. Babu, S.G. Djorgovski, M.J. Graham, D.S. Katz, A. Mahabal, C.D. Miller, R. Nichol, D.E. Vanden Berk, H. Walia, Grist: grid data mining for astronomy, in: Astronomical Data Analysis Software and Systems (ADASS) XIV, October 2004.
[22] A.K. Jain, R.C. Dubes, Algorithms for Clustering Data, Prentice-Hall, Englewood cliffs, NJ, 1988.
[23] I. Janciak, P. Brezany, A. Min Tjoa, Towards the wisdom grid: goals and architecture, in: Proceedings of Fourth International Conference on Parallel Processing and Applied Mathematics PPAM, 2003, pp. 796–803.
[24] R. Jin, G. Agrawal, An efficient implementation of apriori association mining on cluster of SMPS, in: Proceedings of the Workshop on High Performance Data Mining, Held with IPDPS 2001, April 2001.
[25] R. Jin, G. Agrawal, A middleware for developing parallel data mining implementations, in: Proceedings of the First SIAM Conference on Data Mining, April 2001.
[26] R. Jin, G. Agrawal, Shared memory parallelization of data mining (algorithms): techniques, programming interface, and performance, in: Proceedings of the Second SIAM Conference on Data Mining, April 2002.
[27] R. Jin, G. Agrawal, Shared memory parallelization of decision tree construction using a general middleware, in: Proceedings of Europar 2002, August 2002.
[28] R. Jin, G. Agrawal, Communication and memory efficient parallel decision tree construction, in: Proceedings of Third SIAM Conference on Data Mining, May 2003.
[29] R. Jin, G. Agrawal, Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance, IEEE Trans. Knowl. Data Eng. (TKDE) 17 (1) (2005) 71–89.
[30] M.V. Joshi, G. Karypis, V. Kumar, Scalparc: a new scalable and efficient parallel classification algorithm for mining large data sets, in: Proceedings of the International Parallel Processing Symposium, 1998.
[31] G. Kickinger, P. Brezany, A. Tjoa, J. Hofer, Grid knowledge discovery processes and an architecture for their composition, in: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2004), Innsbruck, Austria, February 2004.
[32] R. Machiraju, J. Fowler, D. Thompson, B. Soni, W. Schroeder, EVITA—efficient visualization and interrogation of terascale data sets, in: R.L. Grossman et al. (Eds.), Data Mining for Scientific and Engineering Applications, Kluwer Academic Publishers, Dordrecht, 2001, pp. 257–279.
[33] M. Mehta, R. Agrawal, J. Rissanen, Sliq: a fast scalable classifier for data mining, in: Proceedings of the Fifth International Conference on Extending Database Technology, Avignon, France, 1996.
[34] S. Mehta, K. Hazzard, R. Machiraju, S. Parthasarathy, J. Willkins, Detection and visualization of anomalous structures in molecular dynamics simulation data, in: IEEE Conference on Visualization, 2004.
[35] J. Shafer, R. Agrawal, M. Mehta, SPRINT: a scalable parallel classifier for data mining, in: Proceedings of the 22nd International Conference on Very Large Databases (VLDB), September 1996, pp. 544–555.
[36] V. Stankovski, M. May, J. Franke, A. Schuster, D. McCourt, W. Dubitzky, A service-centric perspective for data mining in complex problem solving environments, in: Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), 2004, pp. 780–787.
[37] D.S. Thompson, R. Machiraju, M. Jiang, V.S. Dusi, J. Nair, G. Craciun, Physics-based mining of computational fluid dynamics data sets, IEEE Comput. Sci. Eng. 4 (3) (2002).
[38] M.J. Zaki, C.-T. Ho, R. Agrawal, Parallel classification for data mining on shared-memory multiprocessors, in: IEEE International Conference on Data Engineering, 1999, pp. 198–205.

Leonid Glimcher recieved his B.S. and M.S. degrees in Computer Science and Engineering from the Ohio State University in 2003 and 2007, respectively. He is currently a Ph.D. candidate in the Computer Science and Engineering Department at the Ohio State University. His research interest include data grid computing, parallel and distributed data analysis, and high-performance computing.

Ruoming Jin is an Assistant Professor in the Department of Computer Science at the Kent State University, Ohio. He received his B.E. and M.E. degrees in Computer Engineering from the Beijing University of Aeronautics and Astronautics, China in 1996, 1999, respectively. He received his M.S. degree in Computer Science from the University of Delaware in 2001 and Ph.D. degree in Computer Science from the Ohio State University in 2005. His research interest includes system support and algorithm design for scalable data mining, data stream processing, massive graph mining, databases and bioinformatics. He has published over 40 research papers in these areas.

Gagan Agrawal is a Professor of Computer Science and Engineering at the Ohio State University. He received his B.Tech degree from Indian Institute of Technology, Kanpur, in 1991, and M.S. and Ph.D degrees from the University of Maryland, College Park, in 1994 and 1996, respectively. His research interests include parallel and distributed computing, compilers, data mining, grid computing, and processing of streaming data. He has published more than 140 refereed papers in these areas. He is a Member of ACM and IEEE Computer Society. He received a National Science Foundation CAREER award in 1998.